# Automated Testing Techniques: Lessons Learned from Prior Automated Testing October 19, 1999 (Version 2.0)

**First Deliverable (FY 1999, Quarter 2) of**

**Automated Testing Techniques**

**Jet Propulsion Lab Center Initiative UPN 323-08-5N**

**Project Officer: Siamak Yassini**

**Prepared by:**

**Martin S. Feather, Richard G. Covington**

**Jet Propulsion Laboratory**

**California Institute of Technology**

**4800 Oak Grove Drive**

**Pasadena CA 91109 USA**

**email: Martin.S.Feather@jpl.nasa.gov, Richard.Covington@jpl.nasa.gov**

**tel: +1 818 354 1194, +1 818 354 6586**

## EXECUTIVE OVERVIEW

This document reports on lessons learned from an earlier investigation of Automated Testing Techniques applied to a mission-critical software component that generates sequences targeted for execution by a highly autonomous spacecraft. Verification and validation of such a class of software presents challenges that are expected to be met by new testing techniques that rely on testing automation. The purpose of the supporting RTOP is to identify, investigate, and evaluate new testing techniques that are automated and reusable, and that can be easily and quickly adapted to the targeted class of software components.

The document is organized as follows. First, background of the investigation and the testing premise is described. Then, the testing process is discussed, including an identification of the major testing challenges. Finally, results of the testing, lessons learned, and concluding observations are presented.

## 1 BACKGROUND AND TESTING PREMISE

The software component chosen as the target of the earlier investigation is the AI planner component of the Remote Agent Experiment (RAX) for the Deep Space 1 (DS-1) spacecraft. The planner takes as input high-level goals, and compiles the goals into lower-level sequences that satisfy the goals while adhering to certain operational restrictions. The resulting sequences will be passed to the spacecraft's intelligent and highly autonomous executive, where they are executed to control major hardware components in achieving the associated goals.

The investigation of automated testing techniques for this software component was focused on the software design team's need for independent verification and validation of the output sequences. Specifically, the goal of the testing was to show that the sequences (1) do not violate operational constraints expressed in the form of spacecraft flight rules, and (2) satisfy the original goals.

It was further assumed that the testing techniques developed would generalize to some appropriate wider class of software components, namely, mission-critical software components with constraints expressible as logical predicates on their states of operation.

## 2 THE TESTING PROCESS AND CHALLENGES

The characteristics of the investigated software, its development environment, and its target execution environment all place special demands on and pose new challenges to existing testing strategies. First, the software in question produces sequences which are highly mission critical, due to the increased level of autonomy of the spacecraft's intelligent executive. Second, again due to the role of autonomy, the sequences produced operate under fewer constraints than traditional sequences, which translates into a larger number of test cases to consider. Finally, the degree of detail desired for each test case, including traceability and justification for the results of the sequence checking, is another dimension that expands the volume of material to be checked. Since the developers depend on the results from early implementation tests as feedback to the ongoing development, short turnaround time for test evaluation is also a critical need. Below is a discussion of the challenges and the strategies expected to meet them in more detail.

**New Testing Challenge (1): Mission-Critical Autonomous Software**
Cost, performance and functionality concerns are driving a trend towards the use of self-sufficient autonomous spacecraft systems that depend less and less on the decisions and inputs of a human controller. They are intended to operate over an extended period, without human intervention or oversight. Since these systems are intended to substitute for human-controlled mechanisms over significant time intervals, it is critical that they function correctly.

Expectation is high that such systems will work well for DS-1, and therefore similar components are planned for use on multiple future spacecraft. As a result, the investigation of the testing technique should include an effort to generalize to a broader class of sequence generators for autonomous spacecraft. Testing should show not only that the component functions correctly for a specific mission, but also that the component will behave correctly in general.

*DS-1 example:* the DS-1 Remote Agent Experiment will be the first artificial intelligence-based autonomy architecture to reside in the flight processor of a spacecraft and *control it for 6 days without ground intervention*.

**New Testing Challenge (2): Number of Test Cases**
Advanced software components also exhibit a much wider range of behaviors than the equivalent mechanisms of more traditional software components. In the environment of traditional sequence generation and testing, the sequence is intended (1) to operate during a relatively brief time period, or (2) to control only one instrument, or both. In the current highly autonomous environment, these restrictions relax or disappear, resulting in a much larger test case space. Developers then face the need to perform more tests to provide adequate coverage of the larger behavior space.

*DS-1 Example:* The DS-1 RAX AI planner exemplifies this testing challenge since it must be crafted to work over a wide range of inputs. This input space is the cross product of the state of the spacecraft at the time that planning is initiated, the set of constraints that correspond to flight rules, and the set of goals that describe the objectives the generated plan is to achieve. In contrast, traditional command sequences generate a much smaller behavioral state space due to the restrictions mentioned above.

**New Testing Challenge (3): Size of Test Cases**
The behaviors of advanced software components can be rich in detail, and are typically structured for processing by other advanced software components. The individual tests themselves can involve considerable detail and become challenging to set up, to run, and to evaluate for correctness.

*DS-1 Example:* The RAX planner yields a plan as a result of a single test run. A typical plan is a detailed and voluminous object, ranging from 1,000 to 5,000 lines long when pretty-printed by Lisp. Furthermore, each such plan must satisfy every one of hundreds of flight rules, and the information to ascertain whether or not a plan satisfies a rule is dispersed through the plan.

**New testing opportunities**
General strategies such as automation are crucial towards achieving the levels of test case coverage needed to meet these challenges. Fortuitously, the nature of advanced software components facilitates the introduction of automation into testing activities. In particular, such components take as input, and yield as output, customized machine-manipulable representations (i.e., in some formal notation other than simply natural language). These machine-manipulable representations are the data on which the automation operates. In contrast, more traditional software components have little between the extremes of requirements expressed in natural language, and the programming language statements in which they are coded.

*DS-1 example:* the RAX planner takes as input flight rules expressed in a formal language (the language of "compatibilities"). Statements in this language take the form of temporal constraints between pairs of actions. Similarly, the RAX planner's output, a plan, is also expressed in a formal language (parameterized actions located on timelines). Both these languages are planner-specific, formal, rigorous notations.

## 3 TESTING RESULTS
In the earlier investigation, automation was successfully applied in the evaluation of test cases for the DS-1 RAX planner. The planner experts developed test cases in the form of goals. The planner was run on each of these test cases to yield a plan or sequence (except for those occasions when the planner failed to produce a plan within the allotted time limit). Automation was introduced by coding or scripting the existing test support software to verify that a plan adheres to all operational constraints (i.e., that it doesn't do anything that it shouldn't), and indirectly that the plan satisfies its original goals (i.e., that it does what it is supposed to). Each of the hundreds of flight rules for the spacecraft is represented in the testing environment by a constraint. The set of constraints is basically "static"

and applies to the checking of any generic plan. But since each plan is generated in accordance with its own unique set of goals, the goals indirectly augment the flight rules set for the specific test case. So, the process of checking the generic flight rules also indirectly checks that the original goals are achievable by the resulting plan. In more formal testing terminology, the automated or scripted constraint checker is called a "test oracle", which is a software application that can ascertain whether the output of a test run is correct

Automation was taken one step further – the test oracles themselves were automatically generated from the set of constraints (flight rules) input to the planner. The net result was a plan checking system that could check the thousands of plans generated in the course of DS-1 testing, and the test system itself could be automatically regenerated whenever the set of planner constraints evolved.

## 4 LESSONS LEARNED

The experience of introducing automation into DS-1 testing was an overall success. However, retrospective study of the experience reveals the following as being areas worthy of further investigation. The RTOP is structured to pursue these investigations in the following months.

**Lesson: Efficiency is *not* always of paramount importance**

The classic notion of a test oracle is of a system that, given a test input and a test output, will answer yes or no to the question "is the output correct for that input". Much literature on test oracles assumes this simple structure, and focuses on making them highly efficient. For example, test oracles are commonly implemented as finite state machines that recognize only violations to the particular input-output relationship they were designed to monitor. The advantage of such implementations is that they scale to huge amounts of information (very voluminous test logs), have "on-line" performance (each additional piece of input is checked in constant time), and consume minimal resources. Such considerations are particularly important if, say, the test oracle is intended to be operational software that resides alongside the running system and monitors that system in real time as a safety check.

In contrast, the DS-1 experience suggests that less efficient oracles may often suffice for a testing environment, where automation may be more important than optimization In contrast to a real time operations environment, the resources consumed by oracles during testing is probably not critical. Likewise, ensuring the fastest possible execution of test oracles is not strictly necessary, since they can be permitted to detect a failed test some time after the test has taken place. Of course, some bounds on run time are a practical need since the developers are expecting to use the test results as feedback into the ongoing development. In the DS-1 case, the test oracles operate faster than the DS-1 planner, that is, they take less time to analyze a plan than the planner takes to generate it. This is not an overly surprising or spectacular result, given that generating a plan is a challenging search problem (hence the whole AI sub-domain of planning), whereas checking a plan once it has been generated is much less complex.

This freedom from the strictest of efficiency concerns allows the consideration of a wider range of analysis techniques. The next lesson learned addresses how to make use of this freedom.

**Lesson: Consider the Cost/Benefit of the Testing Code**

Automation in testing will yield a net benefit only if the development costs of that automation do not exceed the savings obtained from use of the automation.

Our DS-1 experience found the following to be the critical concerns:

- Availability of the domain experts
- Overall development effort (time and budget)

The former refers to the limited availability of the spacecraft planner experts. Hence, the development effort (of the automated generator of automated test oracles) was structured to make best use of their available time. The bulk of the development was performed by an analysis expert who was not a spacecraft planner expert, but who received guidance and assistance from spacecraft planner experts on an as-needed basis.

The latter refers to the usual business case considerations, since project funds paid for the development. Had some alternative, more cost-effective means been available to accomplish the same levels of testing assurance, then those would have been chosen instead.

The freedom from the strict efficiency considerations should be exploited here. Rather than employ sophisticated analysis with a correspondingly tortuous input format, the technique used more mundane analysis whose input format was highly flexible and general purpose. In particular, it included a flexible database engine for analysis. The net result was a smaller gap between project-specific notations (plans and planner constraints) and analysis notations (data and database queries), which reduced the overall effort. In particular, it led to a straightforward representation of plans by the analysis system. One of the spacecraft planner experts was able to make use of this analysis representation directly, in extending the use of the analysis approach.

**Lesson: Test cases have significant structure**

Another characteristic of the classic test oracle is that their output is a simple binary value, either "passed" or "failed". Much research into test oracles includes this fundamental assumption. Yet DS-1 plan checking revealed that there was the need to take into account more of the available structure of the test cases. Correspondingly, test outputs became more than simply "passed" or "failed".

A simple example is that when a plan fails a test, the developers demanded detailed information that traced to requirements, that identified exactly which part of the test had failed, and why. A binary "pass" or "fail" was insufficient.

A subtler example is that of the bounding of DS-1 plans by a beginning time point and an end time point. Some of the constraints (flight rules) spanned such boundaries. Thus the correctness of a plan could depend on the activities before and/or after the boundaries of that plan. The test oracles took these considerations into account, and divided the constraints that the plan was checked against into *three* cases:

- constraints that were satisfied regardless of plan boundaries,
- those that were violated regardless of plan boundaries, and
- those that were satisfied only for certain assumptions about activities outside of those boundaries.

**Lesson: Automated Translation Between Notations is Key**
There is a recurring need to translate between design notation and analysis notation. In order to scale, the translation itself must be automated. Without such automation, a costly manual step would remain, limiting the approach to only the most critical core for which such costs are outweighed by the need for extremely high levels of assurance.

Such manual steps can only rarely be afforded in a typical test plan. For this reason, the DS-1 work included the development of an automatic generator of test oracles. In actuality, this generator was a *translator*, from the planner constraint language to the database query language.

The translator itself was structured as a large procedural program, which in retrospect is easy to see as a bottleneck in the overall development process. The program was hard to maintain even for its creator, not to mention for someone else to extend or adjust. This suggests the need to pay much more attention to the construction of inter-notation translators. Desirable characteristics would include:

- Easily adapted/extended
- Easily understood
- "Obviously" correct

A large procedural program fails all these desires. Two major alternative styles address this problem:

1. Translation based on declarative rules.
2. Translation structured into small composable units.

In the first, the translation is coded as a sequence of language-to-language steps in a declarative style. A generic rule execution mechanism then applies these steps to input, yielding the output step by step. This approach has been studied extensively in the program transformation research community, and commercial tools exist that support such translation activities (e.g., Reasoning Transformation Software Architecture by Reasoning, Inc.).

The second approach emphasizes the overall structure of the translation, encouraging the breakdown of the translation process into many small reconfigurable units, organized into a data-flow like architecture. A parallel RTOP activity is developing a generic mechanism that supports such a structure.

Translation for purposes of test automation generation appears amenable to either approach. The desire for understandability and maintainability favors the second approach. The next phase of this RTOP will consider these issues in greater detail.

**Lesson: Redundant Rationale Useful for Cross-Checks**
The plans generated by the DS-1 planner contain information beyond a simple list of actions arranged on timelines. In particular, they contain traceability information that relates those actions to the constraints that were taken into account in their planning. One of the required capabilities for the test oracles was that they check this information. The planner experts wanted these checks for the additional level of assurance that they would provide. Essentially, traceability checking would increase confidence that the planner was generating correct plans "for the right reasons".

In the event, these checks were also useful to the developer of the test oracle generator, proving effective at finding mistakes in the oracles themselves (or, equivalently, in the code that generated the oracles). For example, the plans contained redundant information that identified for every activity all the constraints that were considered in generating the plan for that activity. The test oracle also had to confirm that the list of constraints that an activity was checked against matched the plan's own constraint list for that activity. The test oracle could not overlook a constraint and still yield a "passed" result.

This "extra information" present in DS-1 plans was recognized to be a useful form of redundant rationale. Redundant information is always useful for cross-checking (especially where the checks are performed automatically, and hence voluminous amounts of information is not an impediment). Furthermore, rationale is a particularly potent form of redundancy. It relates information at different levels of concern. (E.g., DS-1 rationale relates *what* constraints were taken into account with *how* the actions were scheduled so as to satisfy those constraints.)

This study highly recommends that such redundant rationale information be present in system outputs, and that testing encompass the checking of such information.

## 5 CONCLUSION

This report has presented lessons learned based on an earlier investigation of Automated Testing Techniques applied to a mission-critical software component. The main lessons are as follows:

Mission-critical autonomous software offers a new set of testing challenges, notably: the need for thorough testing, handling a large number of test cases, and handling test cases which themselves are large. It also offers a new set of testing opportunities, notably the feasibility of automating significant aspects of testing, stemming from availability of machine-manipulable notations.

Automatic test-case checking (a.k.a. "test oracles") can meet some of those new testing challenges. More surprisingly, maximizing the efficiency of those test oracles is not necessarily of primary importance.

Automatic generation of the test oracles themselves can be beneficial, but is itself a challenging activity, one that warrants further attention.

Testing can make good use of redundancy within the information being checked. Redundant information pertaining to rationale - for example, a trace of the reasoning of why an activity was planned for a given time - is particularly useful. Checking this redundant rationale information extends assurance in the correctness of the system under test (it's doing the right thing and doing it for the right reasons), and in the correctness of the test oracles themselves.

## ACCOMPANIMENTS

Two papers accompany this deliverable:

1. "**V&V of a Spacecraft's Autonomous Planner through Extended Automation**" – produced *prior* to this RTOP, and presented at the **NASA Goddard Software Engineering Laboratory's 23rd Annual Software Engineering Workshop, December 1998**. It outlines the successful use of automation in checking the outputs of DS-1 planner testing.
2. "**Automatic Generation of Test Oracles - From Pilot Studies to Application**" **-** produced as part of this RTOP, and appearing in **Proceedings 4th IEEE International Conference on Automated Software Engineering Conference, Oct. 12-15 1999, Cocoa Beach, Florida. IEEE Computer Society pp 63-72**. It presents some of the lessons learned discussed herein, and provides more details that substantiate those lessons.